

USB/I²C INTE

Paul Goossens



Since the introduction of the I²C bus in the eighties by Philips it has been used many times to provide a simple connection between a PC and home-built circuits. This often made use of a simple interface connected to the parallel port of the PC. This now seems a bit dated, so we've designed a USB/I²C interface for you to build.

Articles about the I²C bus have appeared several times in *Elektor Electronics*. We have even published designs for a whole range of modules that communicated with a PC via the I²C bus. At that time we used an ISA card to provide an I²C interface. Another type of interface that was frequently used was a simple circuit that connected to the parallel port. But with the introduction of modern operating systems it became much more difficult to control the parallel port correctly. The old software would often no longer function properly. We have designed a new I²C interface, which uses the USB bus instead of the parallel port to connect to the PC, so that it can be easily used with newer PCs and modern operating systems.

I²C history

Philips originally developed the I²C bus to provide communication facilities between the various chips found in television, video recorders, etc., without taking up too much space on the PCB. The result is that the I²C bus requires only two signal lines instead of 8 datalines, several address lines and a few control lines, which was the case up to then.

Once this bus was in common use, various other chips were designed with an I²C interface by Philips as well as other manufacturers. Many chips were designed for use in audio and video equipment, such as tone controls, synthesizers, volume controls, and so on. The range has been extended since then and now includes digital I/O buffers, A/D converters etc., giving us a wide range of I²C devices to choose from.

I²C operation

The I²C interface is a so-called master-slave bus. This means that every I²C

bus has one master device, with all other devices acting as slaves. The master is always responsible for supplying the clock signal on the bus. Every transaction may be initiated by the master only. A slave should therefore never start transmitting data on its own accord!

Each slave on the I²C bus has its own unique address. Every transaction begins with a START condition, followed by a 7-bit address and a R/W bit. If a slave with this address is present on the bus it will send an ACK (acknowledge) signal to confirm that it has recognised its address. All the other I²C slaves will now ignore the data on the bus and will only start listening again after a STOP condition has been detected on the bus.

The R/W bit is used to inform the slave whether the master wants to read or write data. When this bit is a '1', the master wants to read data. In this case, the next byte will be put onto the bus by the slave. If this bit is '0' (Write), the following byte(s) will be transmitted by the master. The

master still generates the clock signal in both cases.

At the end of the communication is a STOP condition and the whole process then repeats itself.

Interface

The interface we've designed should satisfy several requirements. We would like to keep the installation under Windows as easy as possible. What's more, it should be (relatively) easy to control the interface. It would also be nice if we could run longer cables between the interface and I²C modules, so the user can connect devices that are a long way away from the PC.

Our first task was to search for a suitable controller with a USB interface. We decided to use a TUSB3410 made by Texas Instruments, as this microcontroller has both a USB as well as an I²C interface. This is described (somewhat misleadingly) as a 'USB to Serial Port Controller'. In fact this is an 8051 compatible controller with a USB inter-

Quick specifications

- **Compact**
- **USB 1.0 full-speed interface**
- **Compatible with USB2.0 hosts**
- **Works with Windows 98SE to Windows XP**
- **Doesn't require separate driver software**
- **The maximum cable length between the interface and modules is at least 100 metres!**
- **I²C frequency: 100 kHz**
- **Simple to use by means of a DLL**
- **Source code for the firmware, DLL and example program is available**
- **Uses a 9 V mains adapter as the power supply**

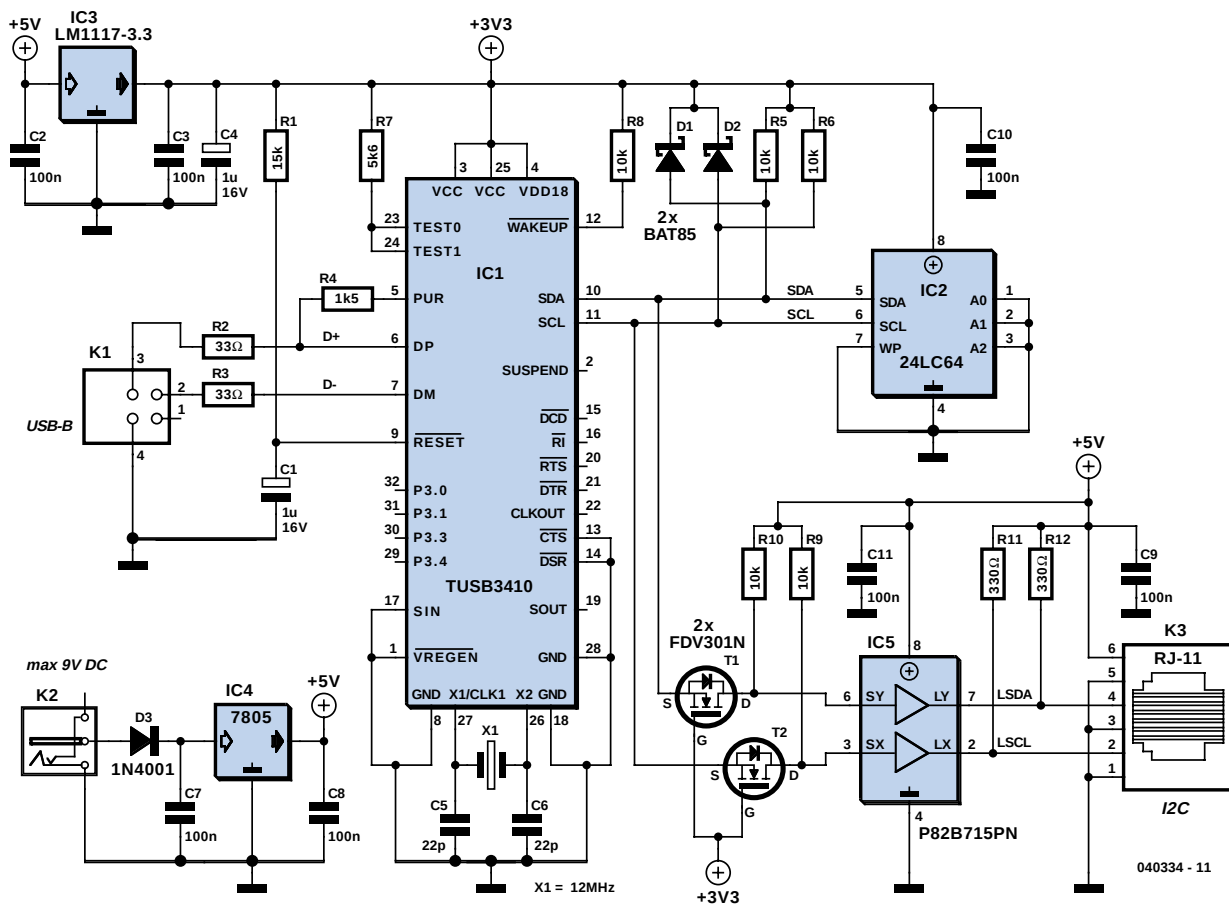


Figure 1. The circuit for the interface is built round a TUSB3410.

face, 16 kBytes of program memory, an enhanced serial port and an I²C interface. All these goodies are contained within a small 32-pin SMD package. To satisfy the requirement for the I²C connection to work over greater distances, we have given the signals an extra boost with the help of a P82B715PN I²C bus extender.

Circuit diagram

The circuit diagram for our interface is shown in **Figure 1**. At the heart of the circuit is the TUSB3410 (IC1) with its associated oscillator (X1, C5 and C6). We've used an external mains adapter for the power supply. Its 9 V output is stabilised down to 5 V by IC4. There is also a requirement for a 3.3 V supply for the processor and the I²C EEPROM. For safety reasons we decided not to supply the circuit with power directly from the USB bus. The 5 V supply is also brought out on this connector and an overload or short could damage the USB interface in the PC.

The data lines from the USB connector (K1) are connected via resistors R2 and R3 to the appropriate data inputs of

the controller. The D+ line is connected via a 1k5 pull-up resistor to the PUR (Pull-Up Resistor) output of IC1. Normally this resistor is connected to the positive supply. This is required for a USB hub to detect that a full-speed USB device is connected to the USB port. By connecting this pull-up resistor to an output of the controller, the controller itself can tell the USB hub that a device is present. The reverse is also possible in that the controller can fool the hub that the device has been disconnected, by forcing this output low. This process is important during the booting of the chip (refer to the inset on booting the TUSB3410). The firmware is stored in an I²C EEPROM (IC2) and is connected to the controller via the I²C bus. Diodes D1 and D2 absorb any potential voltage spikes and protect both I²C signal inputs of the controller. R5 and R6 the pull-up resistors that have to be present on every I²C bus.

You will have noticed that the I²C bus of the controller works at 3.3 V. These signals are not suitable for use in combination with a 5 V chip. Unfortunately, the I²C bus-extender is a 5 V device, so

both I²C signals first have to be converted to a 5 V level. This is taken care of by components T1, T2, R9 and R10. The P82B715PN I²C bus extender along with R11 and R12 allow longer cables to be connected to the I²C bus by lowering the impedance and increasing the current output. These signals are then taken to the outside world via K3.

The double-sided PCB shown in **Figure 3** is very small, but still has all the required connectors on board. Mounting the components shouldn't be very difficult, with the exception of the SMDs: IC1, T1 and T2. It's still possible to solder the two transistors using a soldering iron with a fine tip, but it becomes more difficult with the IC. The IC should first be fixed in the right place on the board with a drop of glue. Then solder all pins together as well as to the solder pads. You should work quickly to avoid the IC becoming too hot. After everything has cooled down again you should place a clean piece of desoldering braid across the pins and remove the excess solder (i.e. the shorts between the pins). Don't attempt this with a solder sucker! Take

care at all times that the IC doesn't overheat. You should finally inspect all pins with a magnifying glass and multi-meter to check that they are connected properly and that there aren't any shorts left.

This completes the trickiest part of the construction.

An example of a suitable module that can be connected to the interface is shown in the **I²C Home Bus** article.

If you prefer to use the I²C bus without the I²C bus-extender, you should make the following modifications:

IC5, R11 and R12 can be left out. Pins 3 and 2 on IC5 have to be connected together, as do pins 6 and 7. This is easily done by putting two short

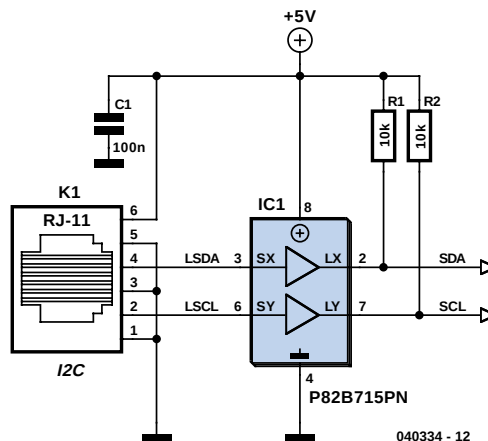


Figure 2. Input circuit for I²C modules when the I²C I/O-extender (IC5) is used.

COMPONENTS LIST

Resistors:

R1 = 15k Ω
 R2,R3 = 33 Ω
 R4 = 1k Ω
 R5,R6,R8,R9,R10 = 10k Ω
 R7 = 5k Ω
 R11,R12 = 330 Ω

Capacitors:

C1,C4 = 1 μ F 16V radial
 C2,C3,C7-C11 = 100nF
 C5,C6 = 22pF

Semiconductors:

D1,D2 = BAT85
 D3 = 1N4001
 IC1 = TUSB3410 (Digikey # 296-12699-ND)
 IC2 = 24LC64, programmed, order code **040334-21**
 IC3 = LM1117-3.3 or LD1117V33C (Digikey # 497-1492-5-ND)
 IC4 = 7805
 IC5 = P82B715PN (Farnell # 559-258; RS Components # 821-784)
 T1,T2 = FDV301N (Farnell # 995-848; RS Components # 354-4907)

Miscellaneous:

K1 = USB-B connector, angled, PCB mount (Farnell # 152-754)
 K2 = mains adapter socket PCB mount
 K3 = 6-way RJ-11 connector (Farnell # 393-8359)
 X1 = 12MHz quartz crystal
 USB cable

PCB, order code **040334-1** (see Readers Services page)

Disk, project software, order code **040334-11** or Free Download

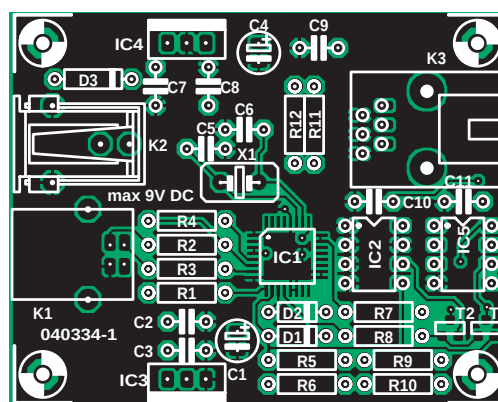
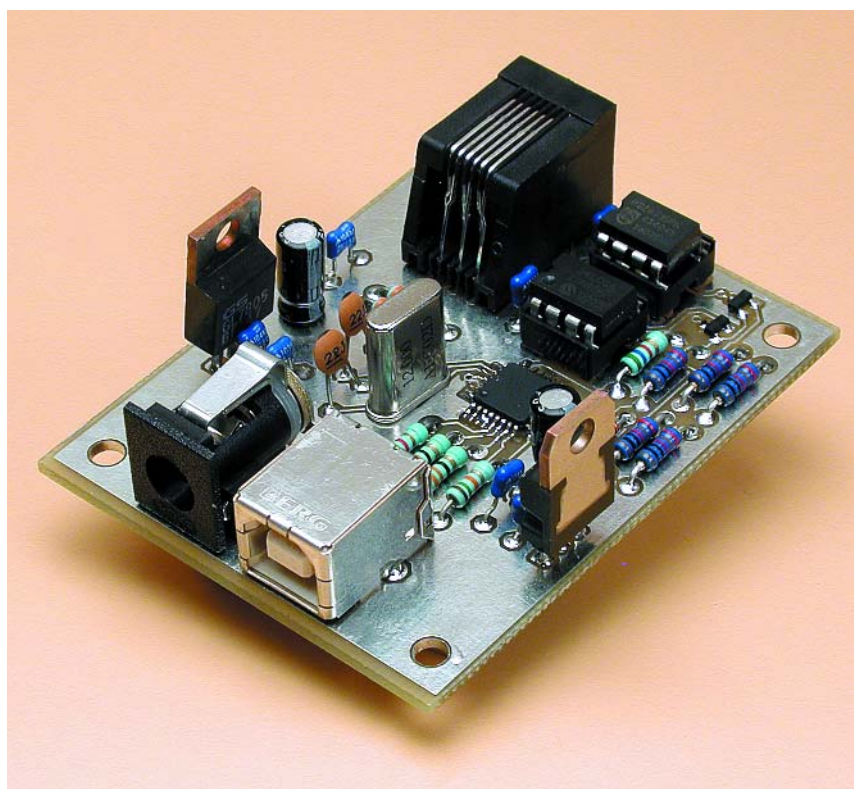


Figure 3. The PCB for the interface. IC1, T1 and T2 are SMDs and have to be soldered carefully using a fine tip. (copper track layout from our website)



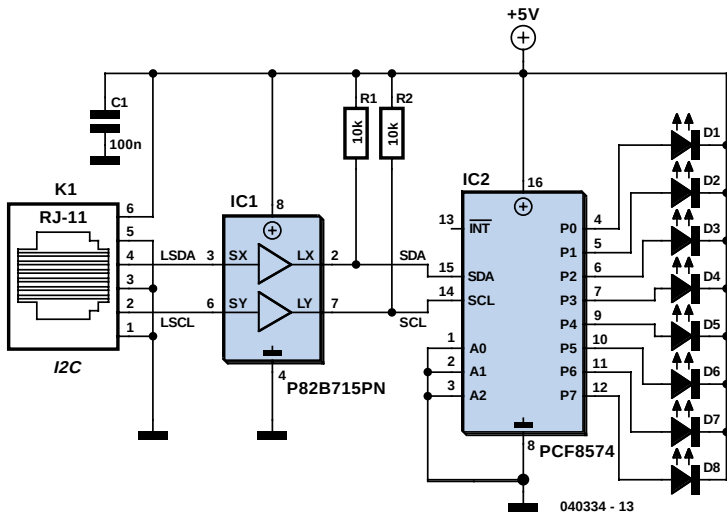


Figure 4. This circuit can be built to help test the interface.

wires into the IC socket. The modules may now connect the SCL and SDA lines directly to the I²C connections of their ICs.

If the I/O extender circuit is included on the PCB, it is essential that all connected modules have the input circuit shown in **Figure 2**.

Installation and DLL

Installing the circuit is fairly straightforward. First you have to connect the mains adapter, then you can connect the circuit to the PC using a USB cable.

Windows (from 98SE onwards) will immediately recognise it as a HID device and after some rattling of the hard drive Windows announces that the device is ready for use.

But there is more to it than that of course. We also need to write a program that makes use of the interface. Its functions obviously depend on the modules that are connected and what you want to do with them. We realise that controlling a HID device isn't exactly a piece of cake for the average programmer. This is why we've made a DLL available which was specially

written to control our USB interface. We would recommend that this DLL is copied to the `C:\Windows\System32` folder. This makes the DLL available to the operating system at all times and you don't have to copy the DLL to the folder where the application using this DLL is stored!

This DLL can be used with any programming language that supports the use of external DLLs. A few examples of such languages are Visual Basic, Visual C++, Delphi and Borland C++ Builder.

Two methods

You can use the DLL from your application in one of two ways. When we wrote the DLL it seemed logical to make the application pass an array of bytes, where the DLL could either store the received bytes or read the bytes to be transmitted.

Most programming languages don't have a problem with the use of arrays as arguments in DLL functions. But it's a bit different with Visual Basic. Although it is possible with this programming language, it is a lot trickier for the novice Visual Basic programmer. For this reason we have included a few extra functions and procedures which make it easier to use the interface under Visual Basic.

To help you on your way we have written two simple examples in Visual Basic and Delphi, which control a standard I²C I/O extender. These examples show the two methods in which the DLL can be used.

The circuit of the module required for the examples is shown in **Figure 4**. The circuit is simple enough for it to be built on a piece of experimenter's board.

Get to work with Word

In the June edition of Elektor Electronics we showed how the well-known word processor MS-Word could also be used to write your own programs. Since most Windows users have a copy of this word processor, it made sense to show you a simple example, written using Word. We have of course kept this example fairly simple and didn't use any arrays when communicating with the DLL!

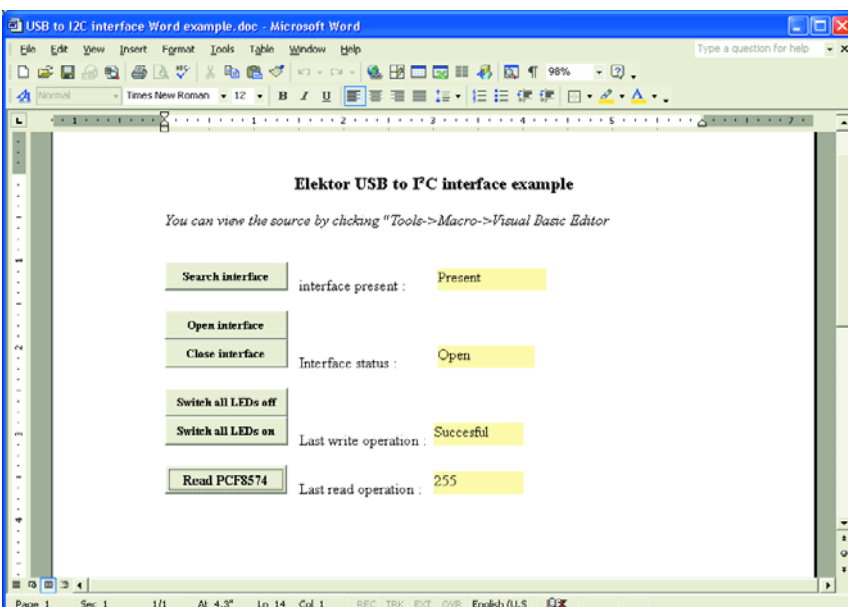


Figure 5. An example program in MS-Word.

The required file is part of the project software (order code **040334-11**) and can be found in the folder 'Word example'. When you open this document you may, depending on the security settings of Word, get a question asking if you want to use macros in this document. You should answer this question with 'yes', otherwise the example won't work.

You should then see the document on the screen as shown in **Figure 5**. When you press the top button the software searches for the interface. The box next to this button shows whether the interface has been found or not.

Before we can use the interface, it must first be opened. This requires a mouse click on the button 'Open interface'. Once you have finished using the interface, it should be closed with a click on the button 'Close interface'. The result of a click on these two buttons can be seen on the associated yellow pane.

The next two buttons are used to turn the LEDs on and off (you have of course built and connected the circuit shown in Figure 4, haven't you?). You can obviously tell from the LEDs on the module if the data was successfully sent, but as you may have gathered, this is also shown in the box to the right of these buttons.

The final button is used to show the status of the inputs of the PCF8574. The value returned by the PCF8574 is shown in the window to the right of this button.

A more detailed explanation is given in the sidebar for 'Using the DLL without arrays'. This sidebar and the source code of the Word document should be enough for a programmer to understand how it all works.

Delphi

Our second example is a very simple application written in Delphi. A question we're sometimes asked is why we use the Delphi language so often in our examples. The reason this programming language is so useful for examples is that it is derived from Pascal and easy to follow. A C programmer should also be able to follow it without problems. The opposite is not true, since a Pascal programmer would have great difficulty in following a C program.

The files for this example can be found in the folder 'Delphi example'. The

DLL functions

Standard:

```
type TReport = array[0..200] of Byte;
function I2C_USB_Present : Boolean; stdcall;
function I2C_USB_Opened : Boolean; stdcall;
function I2C_USB_Open : Boolean; stdcall;
procedure I2C_USB_Close; stdcall;
```

Communication using a program buffer:

```
function I2C_USB_Write (adr : Byte; length : Byte; data : array of Byte) : Boolean;
stdcall;
function I2C_USB_Read (adr : Byte; length : Byte; var data : array of Byte) :
Boolean; stdcall;
```

Communication using the DLL buffer:

```
procedure I2C_USB_ClearWriteBuffer; stdcall;
procedure I2C_USB_ClearReadPointer; stdcall;
procedure I2C_USB_FillBuffer (data: Byte); stdcall;
function I2C_USB_GetBuffer : Byte; stdcall;
function I2C_USB_WriteWithBuffer (adr : Byte) : Boolean; stdcall;
function I2C_USB_ReadWithBuffer (adr:Byte; length:Byte) : Boolean; stdcall;
```

Booting the TUSB3410

The TUSB3410 has a unique way of initialising itself at power-up. In contrast to many of its modern contemporaries, this controller doesn't have on-chip flash memory for storing programs. Instead, it has ordinary RAM for the programs. Consequently, this memory will be completely blank at the start-up. Texas Instruments has included a small program in ROM that is executed at start-up, which cleverly loads a program into RAM. This can happen in two ways.

The boot program first checks if an I²C EEPROM is connected to the I²C bus. If this isn't the case, the controller announces itself on the USB-bus with a certain Vendor-ID and Product-ID. As long as the correct driver has been installed in Windows (freely available from the website of Texas Instruments, <http://www.ti.com>), it will be recognised by Windows as a 'TUSB3410 boot device'. This then causes Windows to send the firmware to the controller. The exact mechanism of this process is explained in the documentation from Texas Instruments. The controller stores the received firmware in its program RAM and then disconnects from the USB bus. The controller then switches to its normal operating mode and starts executing the firmware in the program RAM.

The first method of starting up is the one we've used for our controller. With this method all of the firmware is stored in an I²C EEPROM. The boot program in the controller knows from the header (the first few bytes in the EEPROM) that this EEPROM contains the firmware. Once it has verified that the header is correct, the controller will read the rest of the EEPROM and store it in its program memory. Once that has completed the controller switches to its normal mode and starts executing the firmware. This method has the big advantage that the end user (that's you) doesn't have to install the device driver from Texas Instruments on the PC.

You may well ask what happens when you have two different devices, each using a TUSB3410, but with totally different firmware; how would Windows know what firmware had to be sent to which device? The way round this problem is to provide each USB device with a unique combination of a Vendor-ID and Product-ID. These are stored in the EEPROM and are transmitted by the boot program when it announces itself on the USB bus. The documentation provided by Texas Instruments explains how you can edit the 'inf' file for the TUSB3410 and make it use different firmware for different combinations of the VID and PID.

More information regarding the boot-up process of the TUSB3410 can be found on the website of Texas Instruments.

source code has been kept as compact as possible, which resulted in a single procedure (*Timer1Timer*) for all communication with the interface. This procedure is also responsible for detecting and opening the interface. Immediately afterwards the program reads and writes to the PFC8574, which you had already connected to the I²C bus, of course!

We would like to reassure you that it's not a problem if you open the inter-

face as soon as your program starts and if you close it when the program finishes. In this case we did this in a timer routine that is called twice a second. You can therefore connect and disconnect the USB interface at will while the program runs.

Follow-up

This edition of *Elektor Electronics* also contains an application for this inter-

face, where we can control various peripherals (see 'I²C Home Bus'). It is also possible to design your own modules and to control them using this interface. You will of course need some programming experience under Windows for this.

(040334-1)

Using the DLL

without arrays

In general:

I2C_USB_Open This is called to gain access to the interface. The function returns 'true' when permission is granted.

Reading an I²C chip:

I2C_USB_ClearReadPointer This should be used to initialise a new read command.

I2C_USB_ReadWithBuffer takes care of the reading of bytes via the I²C bus. This function takes two arguments. The first is the I²C address of the device from which we want to read. The second argument specifies how many bytes (maximum of 255) have to be read. This function returns 'true' when the bytes have been successfully read.

The function **I2C_USB_GetBuffer** can then be used to read these bytes one by one.

Writing to an I²C chip:

Before we send any data we should call the procedure **I2C_USB_ClearWriteBuffer**. This procedure makes sure that we begin with a clean slate.

Next we use the procedure **I2C_USB_FillBuffer** to send the data to the DLL one byte at a time. The only argument for this procedure is the byte to be sent. After all bytes of the message have been sent to the DLL we only need to call the function **I2C_USB_WriteWithBuffer** to get these bytes sent via the I²C bus to the desired I²C device. This function takes the address of the receiving I²C device as argument.

with arrays

In general:

I2C_USB_Open This is called to gain access to the interface. The function returns 'true' when permission is granted.

Reading an I²C chip:

In this case we only need one function to read data from the I²C device, which is **I2C_USB_Read**. This function takes three arguments. The first argument specifies the I²C address of the device that we want to communicate with. The second argument states the number of bytes we want to read. The final argument points to a buffer where the received bytes can be stored. This function returns 'true' when the communication went faultlessly.

Writing to an I²C chip:

This command also requires just a single call to the DLL. The function **I2C_USB_Write** takes care of this task. The first argument specifies the I²C address here as well. The second argument specifies how many bytes we want to send. And finally, this function requires a buffer that contains the bytes to be transmitted. Just as with the previous function, the return value tells us if the communication went flawlessly.

N.B. Keep in mind that it's the responsibility of your program to make sure that the buffer is big enough to receive all the data!